EXPLORING EFFICIENT CODING SCHEMES FOR STORING ARBITRARY TREE DATA STRUCTURES IN FLASH MEMORIES

A Senior Scholars Thesis

by

JUSTIN ALLEN FALCK

Submitted to the Office of Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

April 2009

Major: Computer Science

EXPLORING EFFICIENT CODING SCHEMES FOR STORING ARBITRARY TREE DATA STRUCTURES IN FLASH MEMORIES

A Senior Scholars Thesis

by

JUSTIN ALLEN FALCK

Submitted to the Office of Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

Approved by:

Research Advisor: Anxiao Jiang
Associate Dean for Undergraduate Research: Robert C. Webb

April 2009

Major: Computer Science

ABSTRACT

Exploring Efficient Coding Schemes for
Storing Arbitrary Tree Data Structures in Flash Memories.
(April 2009)

Justin Allen Falck
Department of Computer Science
Texas A&M University

Research Advisor: Dr. Anxiao Jiang
Department of Computer Science

Flash memory usage is becoming ever more prevalent in society today. It is being used in everything from portable data storage devices, portable music players, cell phones, and even solid-state (non-mechanical) computer hard drives. Despite their widespread use, there are still many problems inherent to flash memories. Three examples of problems with flash memory are as follows: (1) Block erasures. In flash memories like other memory architectures, cells are organized into large blocks. These cells have charge injected into them individually in order to program the cell. Cells that have charge are then read as “1”, while cells with absence of charge are a “0”. This scheme works well for reading data and writing data once; however, when a cell must be brought back to “0” from “1” the entire large block must be erased and reprogrammed. This reprogramming process is very costly, especially in high write environments (i.e. solid state PC hard drives). (2) Over injection. When charge is injected into a cell to raise its value, it is possible that too much charge will be injected. Over injection can only be

solved by block erasure, which as mentioned before is very costly. (3) Reliability and longevity problems. Block erasures, over injection, and other problems can cause errors in data stored in the memory. In addition, the average flash memory can only endure about 10,000 rewrites before it breaks down. If block erasures are common and not accounted for in some way, flash memory is not a very reliable storage medium. **In this research project, we examine and develop novel coding techniques that work to solve the problem of block erasure**. If block erasures can be minimized, not only will there be an increase in memory speed, but reliability and longevity of the device will increase. We have focused on developing coding techniques for tree data structures because of their wide applicability to computing problems, as well as then using the developed codes as a foundation to generalize to other data structures.

ACKNOWLEDGMENTS

I would like to acknowledge Dr. Anxiao (Andrew) Jiang, Department of Computer Science and Engineering, for his support and mentorship during this research experience. Without his guidance, little of the following work could be accomplished. I would also like to thank the Texas A&M Undergraduate Research Program for allowing me the opportunity to participate in such a fantastic research program. The Undergraduate Research Scholars Program has definitely given me great insight into what a future in Graduate Studies entails. Finally, I would like to thank my wonderful fiancée Ashley. It was her who inspired me to participate in this program, and she helped support me through it every step of the way.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Flash memories have become the most important type of non-volatile memory (NVM) due to their high performance and physical robustness. They are widely used in mobile, embedded and mass storage systems, including cell phones, sensors, computers, etc. [1]

Flash memory usage is becoming ever more prevalent in society today. It is being used in everything from portable data storage devices, portable music players, cell phones, and even solid-state (non-mechanical) computer hard drives. Despite their widespread use, there are still many problems inherent to flash memories. Three examples of problems with flash memory are as follows: (1) Block erasures. In flash memories like other memory architectures, cells are organized into large blocks. These cells have charge injected into them individually in order to program the cell. Cells that have charge are then read as “1”, while cells with absence of charge are a “0”. This scheme works well for reading data and writing data once; however, when a cell must be brought back to “0” from “1” the entire large block must be erased and reprogrammed. This reprogramming process is very costly, especially in high write environments (i.e. solid

---

This thesis follows the style of *IEEE Transactions on Information Theory*.

state PC hard drives). (2) Over injection. When charge is injected into a cell to raise its value, it is possible that too much charge will be injected. Over injection can only be solved by block erasure, which as mentioned before is very costly. (3) Reliability and longevity problems. Block erasures, over injection, and other problems can cause errors in data. In addition, the average flash memory can only endure about 10,000 rewrites before it breaks down. If block erasures are common and not accounted for in some way, flash memory is not a very reliable storage medium. If block erasures can be minimized, not only will there be an increase in memory speed, but reliability and longevity of the device will increase.

The objective of this research project is to explore novel theories for data storage in flash memories. In particular, the focus is on storage techniques that tolerate support efficient data rewriting **(minimizing block erasures).** This research project explores the following area:

Study Efficient Data Rewriting Techniques

As computers of today gain faster processors, more RAM, and better video processing capabilities, the hard disk storage medium has remained mostly unchanged. Currently, the magnetic hard disk is the only mechanical component in the modern computer and serves as a severe bottleneck when compared to the rest of the PC's operating speeds. Flash memory could serve to remedy this situation if used as a solid state hard drive.

Creating efficient solid state hard drives is a problem currently because modifying data is extremely difficult. When rewriting data, there is a large chance that some cells will be taken from “1” to “0” which requires a block erasure. This study will look at novel coding schemes that work to maximize the available number of rewrites before having to perform a block erasure. To complete this task, a one-to-many mapping will be created between information data and cells. To modify data, we only increase cells values; therefore eliminating the problem of taking a “1” to a “0”. As long as a “1” is not required to go to “0” when writing, no block erasure is necessary. This is made possible by the one-to-many mapping, and such a rewriting process ends only when the cell levels reach the maximum value, at which point a block erasure becomes necessary. The objective of the project is to maximize the number of data rewrites, while keeping storage density as high as possible. By maximizing the number of rewrites, flash memory reliability is also increasing.

The area above will be studied for a specific case. In the project, we will consider coding schemes that store arbitrary tree data structures in flash memories. Because the tree is one of the more complex and important structures in computing, it is an obvious place to begin. Any coding techniques developed for this case should be generalized to broader classes of data structures.

CHAPTER II

METHODS

To begin studying novel theories for data storage in flash memories, one must find a place to begin. Beginning by studying coding methods for arbitrary data structures is a problem that is very difficult to define clearly. So, to begin the study, we looked at novel ways to store tree structures in flash memories. Of the 3 main problems outlined in the introduction, the one we have most focused on to this point is solving **block erasure**. Block erasure is very costly, and can be alleviated to an extent through maximizing the number of cell rewrites in a given block.

Storing full binary trees

Full binary trees are the simplest tree structures to study. The full binary trees consist of a root node that must have exactly two nodes as “children”. Each successive node must also have exactly two children. Figure 1 shows an example of a full binary tree, the topmost node being the root. These structures are countable and very predictable. We started by looking a very simple, naïve approaches for storing full binary trees in flash memory to gain an understanding of how the memory cells function.

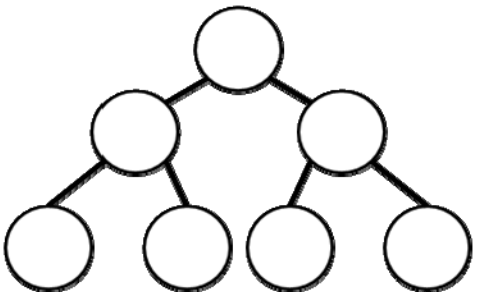


Fig. 1. An Example of a Full Binary Tree.

The first naïve attempt was simply storing the data in a one dimensional array of *n* cells with *q* levels. In this method, when q is at an even level the cell is read as a zero and when q is at an odd level the cell is read as a one. The cells are then indexed in such a way that the tree structure can be discerned (See Figure 2). We realized quickly that this method has a very common pathological case. When one particular cell is added/deleted in rapid succession, which is common in computing applications, it quickly reaches the maximum q value which can only be brought back down to the minimum q through block erasure. This method turned out not to be very optimal due to this pathological case.

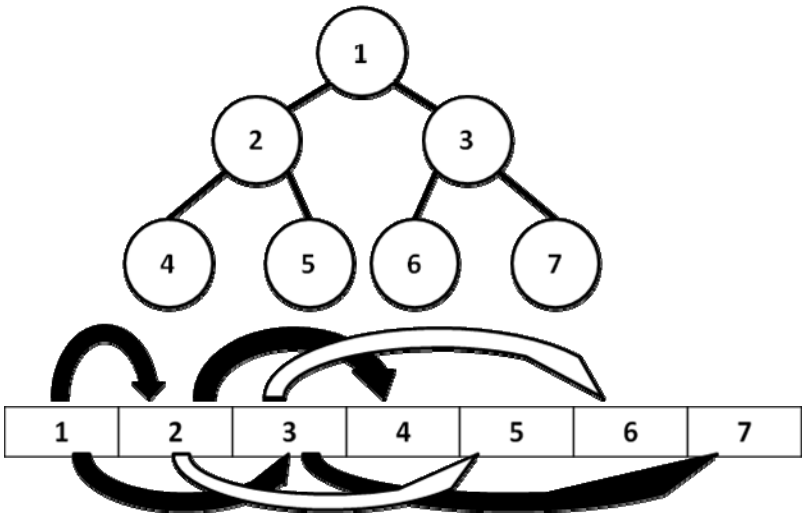


Fig. 2. Example of How to Use a One-dimensional Array to Store a Full Binary Tree.

Having a better understanding of how the cells and errors work, the next attempt was more sophisticated. Realizing that full binary trees are countable, we completed some research and found that full binary trees with *n* internal nodes, nodes with at least one child, are counted by the Catalan Numbers derived from combinatorics. The Catalan numbers, *C(n),* are generated by the following expression:

$$C(n) = \frac{2n!}{n!(n+1)!} \qquad (1)$$

Recalling a past research experience, there is a proof that maps each full binary tree with *n* internal nodes to a unique sequence of integers of length *n* called a *parking function*.[2] Parking functions are defined in the following way:

$$a_1, a_2, a_3, ..., a_i \quad \textit{where} \quad a_i \le i \qquad (2)$$

This proof also depends on knowledge of a concept called Dyck Paths. Dyck Paths consist of all paths through a lattice structure taking only steps (1, 0) and (0, 1) and remaining less than or equal to the line y = x. Figure 3 shows an example of Dyck Paths.

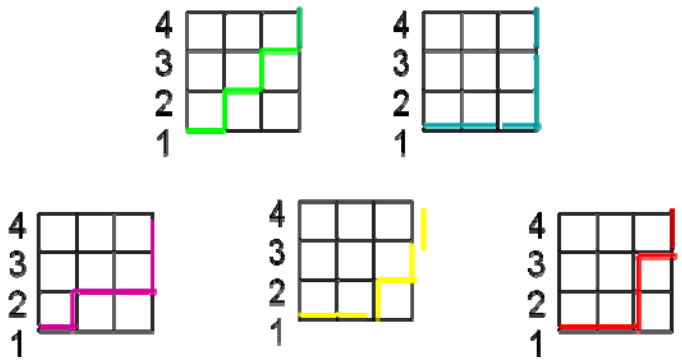


Fig. 3. Example of Dyck Paths for n = 3.

To prove the mapping, using a depth first traversal of a given tree, assign “1” for every leaf node, and a “0” for every internal node. This string of ones and zeroes corresponds to Dyck paths by taking “1” as a (1, 0) step and a “0” as a (0, 1) step. Parking functions map to Dyck paths by taking the level of each horizontal step. In Figure 3, the top-left Dyck path represents the parking function (1, 2, 3). Since there is a bijection between full binary trees and Dyck paths, and full binary trees and parking functions, there is a bijection between parking functions and full binary trees. [2] Having the trees identified by these relatively short integer sequences, we went one step further. Using a dynamic programming algorithm, we were able to reduce these integer sequences to a single

integer index. To do this, the algorithm orders the parking functions into subgroups based on the last integer, then further orders the subgroups into smaller groups based on the second to last integer and so forth until the first integer in the sequences is reached (Reference Table 1).

TABLE I

Indexing Parking Functions of Length 3

Parking Function	Index Value
111	**1**
112	**2**
122	**3**
113	**4**
123	**5**

In effect, by storing the number of internal nodes and the index, all the relevant information regarding a particular full binary tree is represented. Using a mapping that stored these two values while load-balancing the cell usage, would maximize the number of rewrites for this case.

Moving to full N-ary trees

The next step in the process of studying theories for data storage in flash memories is to generalize the data structure we are looking at by one level. Instead of looking at storing

simply full binary trees, we started looking at how to store full trees of any degree (i.e. binary, ternary, etc.). Going back to the full binary trees, we used Catalan Numbers in that case to index and store the various trees. We discovered that there is a generalized formula for the Catalan Numbers called the Fuss-Catalan Numbers that works in the same way as previously seen with the addition of a parameter *p* for the degree of the tree.

$$ {}_pC_k = \frac{1}{(p-1)(k+1)}\binom{pk}{k} \qquad (3) $$

Through extension, the previous proof techniques and dynamic programming algorithm worked to effectively index all full N-ary trees.

Insertion and deletion: unbalanced N-ary trees

While balanced trees are easy to analyze and understand, they are not the most common type of tree structure. The number of unbalanced trees is significantly larger than balanced trees and when inserting and deleting data from a tree one node at a time unbalanced trees are going to result. We hit a setback here due to the fact that the above indexing scheme described only full trees. Although there was no available mechanism to index unbalanced trees efficiently, we were able to use the information in the above method to lead in a promising direction. Catalan numbers also count a particular set of bit strings which were used in the mapping from trees to parking functions above. These bit strings are useful not only in mapping trees to parking functions, but are useful in storing the tree data as they are. These strings are known as planar codes and are unique

to each tree structure. Using $2n-2$ total bits, there is a planar code to represent the tree with n nodes. This discovery allowed us to develop some interesting results.

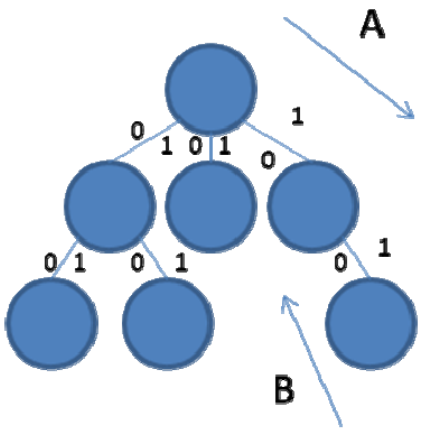
CHAPTER III

RESULTS

Once we realized how efficient planar codes are at representing arbitrary trees, we used them as the foundation for two efficient coding schemes for trees that support insertion/deletion of nodes.

Planar codes

Planar codes allow for representation of a tree as a single bit string of length 2n-2 where n is the number of nodes. [3] Figure 4 illustrates the method for finding the planar code of an arbitrary tree.



Planar codes are generated in the following way:

1. Starting at the root node, begin traversing edges with the right most child, following the arrow labeled A.
2. Assign a "1" for each step away from the root and a "0" for each step toward the root. (Following the B arrow back up toward the root)
3. Continue this method until all edges are counted for both a "1" and a "0".
4. Always traverse the right most edges first, and continue as deep as possible before returning toward the root.

Fig. 4. Generating Planar Codes. The planar code for this example tree is "110010110100".

Insertion and deletion of nodes using planar code representation is straightforward. All one must do to insert a node is to insert a "10" pair at any location in the bit string. To delete a node or branch, all one must do is match a "1" and a "0" and remove those bits and all other bits contained between them.

Multidimensional flash codes

The research of using coding to maximize rewrites and minimize block erasures in flash memories has received lots of attention in recent years. [4] [5] In this study, we first use the multidimensional flash codes proposed in [6]. This code supports both insertion and deletion of nodes into an arbitrary tree and works by using more bits than necessary to store the planar code. Figure 5 demonstrates a small example of how this coding scheme works. The repetition code works by choosing an arbitrary number of cells to represent one bit of data. In the figure, we have chosen nine cells for each bit. These cells are then partitioned as shown in the example into two dimensional areas. The information is stored in the parity of ones and zeroes of the partition. For example, if the partition for bit 3 has one “1” in it, the parity of ones is odd and the bit represented is a “1”. If the parity of ones in the partition is even, then the bit is a “0”. In this two dimensional example, 4 bits of data are stored. The scheme can be extended to support any number of bits in the following way:

$$2^D = K \qquad (4)$$

where D is the dimension of the cell space, and K is the number of bits represented.

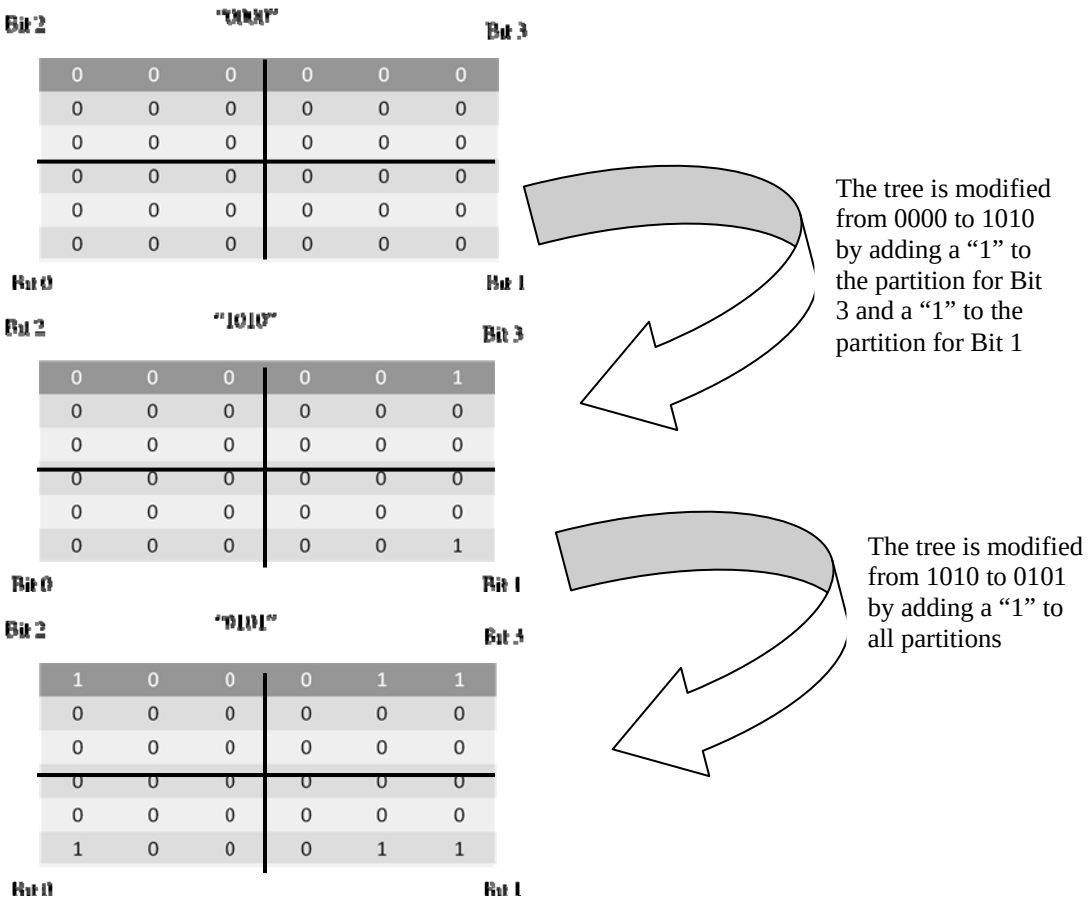


Fig. 5. Multidimensional Flash Code Example.

The repetition code appears to be a sound scheme for dealing with the problem of block erasures. The scheme allows for numerous rewrites to data without having to erase blocks, is easily extended to deal with arbitrary code lengths, and is straightforward to implement and use. However, the method does have very large memory overhead. The flash memory may gain read/write speed and may become more durable due to less block erasures, but this method requires a large amount of free cells to implement, which lowers the free space for data storage.

Edge traversal code

This method keeps track of modifications made to the tree using a transition graph, and only updates a global store of the complete tree at an arbitrary, user-defined interval. A more efficient coding scheme is used in this case than in the simple repetition outlined before. The global store simply stores the whole planar code in one block, and then the subsequent blocks store the edge traversed in the insertion transition graph. To enable rewrites while minimizing block erasures, the edge values are stored in following way, where K is the total number of edge values minus 1, i = K, and C_i is the binary value of the cell at index i: $\sum_{i=1}^{K} iC_i \bmod K + 1$ (5)

Figure 6 shows an example of how the edge traversal scheme would work.

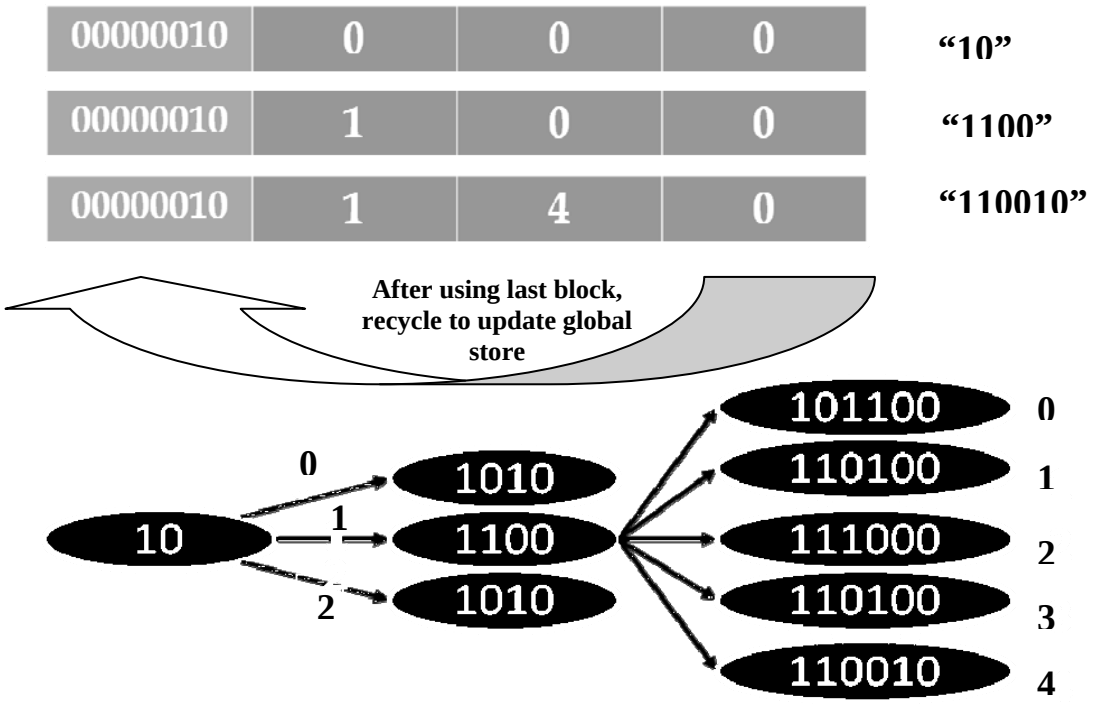


Fig. 6. Example of Edge Transition Scheme.

This edge traversal method appears to be the more efficient of the two schemes. The scheme allows for insertion/deletion modifications to arbitrary tree structures while maintaining a high level of rewrite ability. This method also does not have the tremendous memory overhead of the repetition coding scheme.

CHAPTER IV

SUMMARY AND CONCLUSIONS

As this project is completed, there is still much work to be done. Research will continue in this area for the foreseeable future as flash memory becomes ever more important in everyday life. This thesis explores various methods for efficiently storing tree data structures in flash memory, while minimizing block erasures. Of the two schemes presented, the edge traversal scheme appears to have the most promise. While the repetition scheme is straightforward to implement and allows for a large number of rewrites, it introduces the problem of the high memory overhead. The edge traversal scheme needs to be explored further and perhaps generalized for other data structures such as graphs.

These schemes were designed to solve one type of error for one type of data structure. Other schemes will need to be developed for other data structures and errors, hopefully leading to a universal scheme that solves all of the problems of flash memory efficiently.

REFERENCES

[1] P. Cappelletti, C. Golla, P. Olivo and E. Zanoni (Ed.), *Flash Memories* 1st Edition, Boston: Kluwer Academic Publishers, 1999.

[2] C. H. Yan and J. Falck, Combinatorial Statistics on Parking Functions, Texas A&M Univ. Research Experience for Undergraduates, 2007.

[3] L. Lovász, J. Pelikán and K. Vesztergombi, *Discrete Mathematics Elementary and Beyond* 1st Edition, New York: Springer, 2003.

[4] A. Jiang, V. Bohossian and J. Bruck, "Floating Codes for Joint Information Storage in Write Asymmetric Memories," in *Proc. of the IEEE International Symposium on Information Theory* (ISIT'07), 2007, pp. 1166-1170.

[5] A. Jiang and J. Bruck, "Joint Coding for Flash Memory Storage," in *Proc. IEEE International Symposium on Information Theory* (ISIT'08), pp. 1741-1745, July 2008.

[6] E. Yaakobi, A. Vardy, P. H. Siegel and J. K. Wolf, "Multidimensional Flash Codes," *Annual Allerton Conference on Communication, Control and Computing*, Monticello, Illinois 2008.

CONTACT INFORMATION

Name:	Justin Allen Falck
Professional Address:	c/o Dr. Anxiao (Andrew) Jiang Department of Computer Science HRBB 427B Texas A&M University College Station, TX 77843
Email Address:	justin.falck@gmail.com
Education:	B.S., Computer Science, Texas A&M University, May 2009 Undergraduate Research Scholar